



图文 37 基于MQ实现订单系统的核心流程异步化改造，性能优化

638 人次阅读 2019-11-25 07:00:00

[详情](#) [评论](#)

基于MQ实现订单系统的核心流程异步化改造，性能优化完成！

石杉老哥重磅力作：《互联网java工程师面试突击》（第3季）【强烈推荐】：



全程真题驱动，精研Java面试中6大专题的高频考点，从面试官的角度剖析面试

(点击下方蓝字试听)

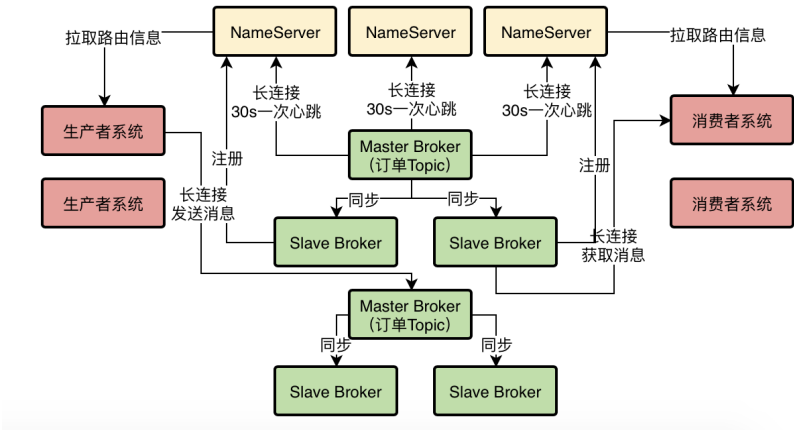
[《互联网Java工程师面试突击》（第3季）](#)

1、万事俱备只欠东风：开始做项目！

经过之前一段时间的学习、忙活以及折腾，小猛终于把RocketMQ的核心架构原理，还有小规模集群的部署和压测，以及最终生产环境的集群部署，全部都搞定了

目前小猛手头已经有了一套3台NameServer机器+6台Broker机器的生产集群，而且对集群的生产参数都进行了适当优化，足以抗下每秒十多万消息的处理。

小猛满意的盯着自己电脑屏幕上的一个生产部署架构图，回顾了一下自己的RocketMQ集群部署情况。



现在既然已经有了一套MQ生产集群了，那下一步当然是基于MQ开始改造订单系统架构了！

应该全面在订单系统的各个环节引入MQ技术，来解决订单系统目前面临的各种技术问题，全面优化订单系统的各项指标！

2、从哪里开始入手改造订单系统？

小猛接着就开始思考了，之前跟明哥已经分析了很多订单系统目前面临的技术问题，到底应该从哪个点入手开始引入MQ技术进行架构改造呢？

针对这个问题，小猛再一次请教了一下明哥。

明哥听到这个问题之后，给小猛分析了一下自己的看法。目前订单系统面临的技术问题包括以下一些环节：

- 下单核心流程环节太多，性能较差
- 订单退款的流程可能面临退款失败的风险
- 关闭过期订单的时候，存在扫描大量订单数据的问题
- 跟第三方物流系统耦合在一起，性能存在抖动的风险
- 大数据团队要获取订单数据，存在不规范直接查询订单数据库的问题
- 做秒杀活动时订单数据库压力过大

针对这些问题，实际上比较合适的就是从第一个问题开始解决，因为下单流程性能较差是目前比较明显的问题，而且是比较严重影响用户体验的。而订单退款失败这种是属于小概率出现的问题，即使出现也可以通过人工处理给解决。

至于关闭过期订单存在大量订单数据扫描的问题，这个问题目前凸显还不严重，因为目前订单数据量还没有那么大。跟第三方物流系统的耦合导致系统性能抖动，也是小概率出现的，并不是经常出现的。

而大数据团队直接查订单数据库跑报表出来，目前压力有点大，但是还不会对订单库造成过大的影响。

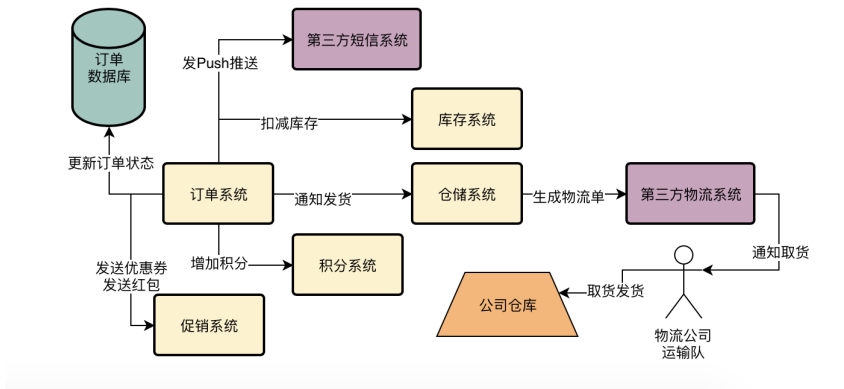
至于秒杀时订单数据库压力过大，也不是目前的主要问题，因为秒杀活动不是经常有，而且目前即使压力过大，但是MySQL部署在高配置物理机上，基本上也能抗住的。

所以经过上述分析过后，明哥的建议是，从下单核心流程开始引入MQ技术进行改造，然后逐步解决订单退款失败问题、跟第三方物流系统耦合导致的性能抖动问题、大数据团队直接查询订单库的问题、秒杀活动时订单库压力过大的问题、关闭订单时扫描大量订单数据的问题。

3、技术方案：通过引入MQ实现订单核心流程的异步化改造

听完明哥的分析之后，小猛接下来就开始进入第一步，尝试在订单系统中引入MQ技术来实现订单核心流程中的部分环节的异步化改造了。

首先小猛先回顾了一下支付订单的核心流程，如下图所示。



现在每次支付完一个订单后，都需要执行一系列的动作，包括：

更新订单状态
扣减库存
增加积分
发优惠券
发短信
通知发货

这会导致一次核心链路执行时间过长，可能长达好几秒种。

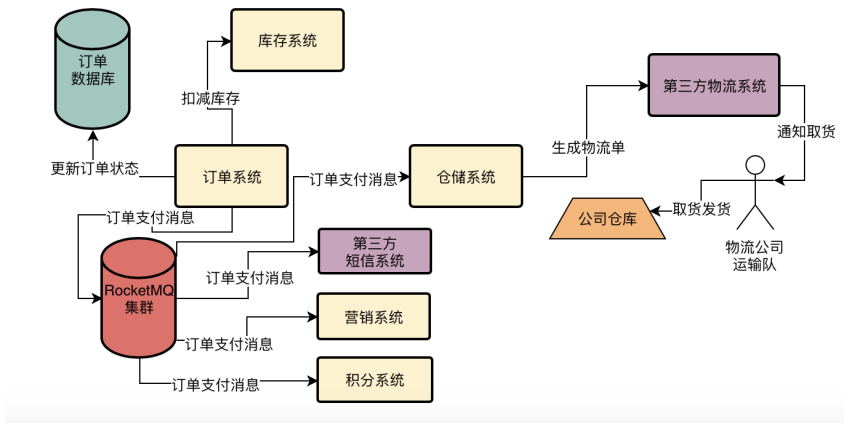
不知道大家是否遇到过一些APP，有时候你下单过后跳转到第三方支付界面（比如支付宝或者微信），然后等你成功支付过后，退回到APP自己的界面上

此时APP上会显示一个圆圈不停的旋转，提醒你等待几秒钟让后台确认订单处理成功，这个等待的过程如果时间较长，往往对用户体验是很不好的。

所以实际上我们需要的一个效果是：在用户支付完毕后，只要执行最核心的更新订单状态以及扣减库存就可以了，保证速度足够快。

然后诸如增加积分、发送优惠券、发送短信、通知发货的操作，都可以通过MQ实现异步化执行。

因此在这个思路指导下，小猛画出了一个订单核心流程的改造图。



在上面的图里，订单系统仅仅会同步执行更新订单状态和扣减库存两个最关键的操作，因为一旦你支付成功，只要保证订单状态变为“已支付”，库存扣减掉，就可以保证核心数据不错乱。

然后订单系统接着会发送一个订单支付的消息到RocketMQ中去，然后积分系统会从RocketMQ里获取到消息，然后根据消息去累加积分

营销系统会从RocketMQ里获取到消息然后发送优惠券，推送系统会从RocketMQ里获取到消息然后推送短信，仓储系统会从RocketMQ里获取到消息然后生产物流单核和发货单，去通知仓库管理员打包商品，准备交接给物流公司去发货。

在上面的改造后的架构中，我们可以举个例子来计算一下引入MQ对订单核心流程的性能优化的效果。

比如更新订单状态需要耗费30ms，调用库存服务的接口进行库存扣减需要耗费80ms，增加积分需要耗费50ms，派发优惠券需要耗费60ms，发送短信需要耗费100ms（涉及与第三方短信系统交互，可能性能抖动会达到1秒+），通知发货需要耗费500ms（因为涉及到跟第三方物流系统交互以及与仓库管理系统交互，比较耗费时间，而且可能会性能抖动达到1秒+）。

如果没有进行架构改造，每次支付成功后都需要由订单系统调用大量的其他系统进行各种操作，可能一次订单核心链路的执行需要接近1秒钟

而且如果第三方短信系统以及第三方物流系统出现性能抖动，那么可能一次核心流程就要几秒钟。

但是现在经过上述改造过后，一旦你支付成功，实际上订单系统只需要更新订单状态（30ms）+扣减库存（80ms）+发送订单消息到RocketMQ（10ms），一共120ms就可以了

对于终端用户而言，一旦支付成功退回到APP界面，还没等你反应过来，可能就显示给你订单支付成功的界面了，不会出现一个圆圈不停的旋转提醒你等待后台检查订单是否支付成功。

而积分系统、营销系统、推送系统、仓储系统都会自己从RocketMQ里去获取订单支付消息执行自己要处理的业务逻辑，不会再影响订单核心链路的性能。

4、在订单系统中如何发送消息到RocketMQ？

在小猛设计完上述方案之后，就要开始落地实施这个技术方案了，这里就涉及到了两个部分

一个是订单系统自身的改造，他需要去除掉调用积分系统、营销系统、推送系统以及仓储系统的逻辑，而改成发送一个订单支付消息到RocketMQ里去；

另外一个积分系统、营销系统、推送系统以及仓储系统的改造，需要从RocketMQ里获取消息，然后根据消息执行自己的业务逻辑。

因此首先我们给大家看一个代码示例，比如原来的订单支付成功的接口如下所示：

```
/**
 * 收到订单支付成功的通知
 */
public void payOrderSuccess(Order order) {
    updateOrderStatus(order); // 更新本地订单数据库里的订单状态
    stockService.updateProductStock(order); // 调用库存服务的接口，扣减库存
    credieService.updateCredit(order); // 调用积分服务的接口，增加积分
    marketingService.addVoucher(order); // 调用营销服务的接口，增加优惠券
    pushService.sendMessage(order); // 调用推送服务的接口，发送短信
    warehouseService.deliveryGoods(order); // 调用仓储服务的接口，通知发货
}
```

现在的话，则需要对上述代码做一个改造，去除掉一些代码逻辑，然后增加一个发送消息到RocketMQ的代码逻辑。

如果要发送消息到RocketMQ，则首先需要在项目里引入下面的依赖：

```
<dependency>
    <groupId>org.apache.rocketmq</groupId>
    <artifactId>rocketmq-client</artifactId>
    <version>4.3.0</version>
</dependency>
```

接着我们需要封装如下的一个RocketMQ生产者的类，类很简单，具体类的注释都写在下面了，大家看一下类的注释就知道是怎么用的。

```

public class RocketMQProducer {

    // 这个是RocketMQ的生产者类，用这个就可以发送消息到RocketMQ

    private static DefaultMQProducer producer;

    static {

        // 这里就是构建一个Producer实例对象

        producer = new DefaultMQProducer("order_producer_group");

        // 这个是为Producer设置NameServer的地址，让他可以拉取路由信息

        // 这样才知道每个Topic的数据分散在哪些Broker机器上

        // 然后才可以把消息发送到Broker上去

        producer.setNamesrvAddr("localhost:9876");

        // 这里是启动一个Producer

        producer.start();

    }

    public static void send(String topic, String message) throws Exception {

        // 这里是构建一条消息对象

        Message msg = new Message(

            topic, // 这就是指定发送消息到哪个Topic上去

            "", // 这是消息的Tag，我们后续再讲

            message.getBytes(RemotingHelper.DEFAULT_CHARSET) // 这是消息

        );

        // 利用Producer发送消息

        SendResult sendResult = producer.send(msg);

        System.out.printf("%s\n", sendResult);

    }

}

```

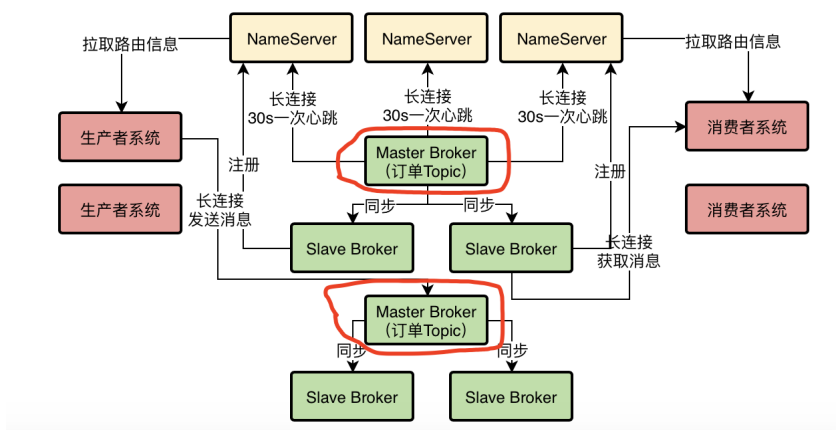
通过上述代码就可以让订单系统把订单消息发送到RocketMQ的一个Topic里去了。

5、订单消息会进入哪个Broker里去呢？

那么大家肯定会疑惑了，按照我们部署的MQ集群而言，Master Broker有两台，那么此时消息会进入哪个Master Broker里去呢？

实际上我们之前说过，Topic是一个逻辑上的概念，实际上他的数据是分布式存储在多个Master Broker中的

如下图所示



我们可以看到图里两个红圈，意思就是“TopicOrderPaySuccess”这个Topic的数据会分散在两个Broker中。

因此当你发送一个订单消息过去的时候，会根据一定的负载均衡算法和容错算法把消息发送到一个Broker中去。

当然肯定很多朋友会问了，那么Topic的数据到底是分散在几个Broker上？可以分散在多少个Broker上？Producer到底是如何选择Broker发送消息过去的？

这些问题大家别着急，现阶段先了解到这个程度就行，后面专栏里有很多RocketMQ底层实现机制的分析，到时候这些问题都会迎刃而解的。

6、其他系统改造为从RocketMQ中获取订单消息

接着下一步就要推动积分系统、营销系统、推送系统、仓储系统的负责人在自己的系统里改造为从RocketMQ中去获取订单消息，然后根据获取到的消息执行对应的业务逻辑

因此小猛给出了一段示例性的从RocketMQ中消费消息的代码。

```
public class RocketMQConsumer {

    public static void start() {
        new Thread() {

            public void run() {
                try {
                    // 这是RocketMQ消费者实例对象
                    // "credit_group"之类的就是消费者分组
                    // 一般来说比如积分系统就用"credit_consumer_group"
                    // 比如营销系统就用"marketing_consumer_group"
                    // 以此类推，不同的系统给自己取不同的消费组名字
                    DefaultMQPushConsumer consumer =
                        new DefaultMQPushConsumer( consumerGroup: "credit_group");
                    // 这是给消费者设置NameServer的地址
                    // 这样就可以拉取到路由信息，知道Topic的数据在哪些broker上
                    // 然后可以从对应的broker上拉取数据
                    consumer.setNamesrvAddr("localhost:9876");

                    // 选择订阅"TopicOrderPaySuccess"的消息
                    // 这样会从这个Topic的broker机器上拉取订单消息过来
                    consumer.subscribe( topic: "TopicOrderPaySuccess", subExpression: "*");

                    // 注册消息监听器来处理拉取到的订单消息
                    // 如果consumer拉取到了订单消息，就会回调这个方法教你处理
                    consumer.registerMessageListener(new MessageListenerConcurrently() {

                        public ConsumeConcurrentlyStatus consumeMessage(
                            List<MessageExt> msgs,
                            ConsumeConcurrentlyContext context) {
                            // 在这里对获取到的msgs订单消息进行处理
                            // 比如增加积分、发送优惠券、通知发货，等等
                            return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
                        }

                    });

                    // 启动消费者实例
                    consumer.start();
                    System.out.printf("Consumer Started.%n");

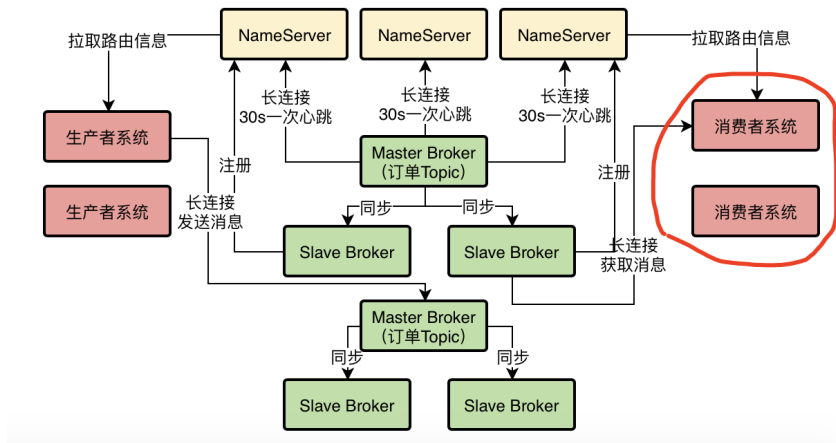
                    while(true) { // 别让线程退出，就让创建好的consumer不停消费数据
                        Thread.sleep( millis: 1000);
                    }

                } catch (Exception e) {
                    e.printStackTrace();
                }
            }

        }.start();
    }
}
```

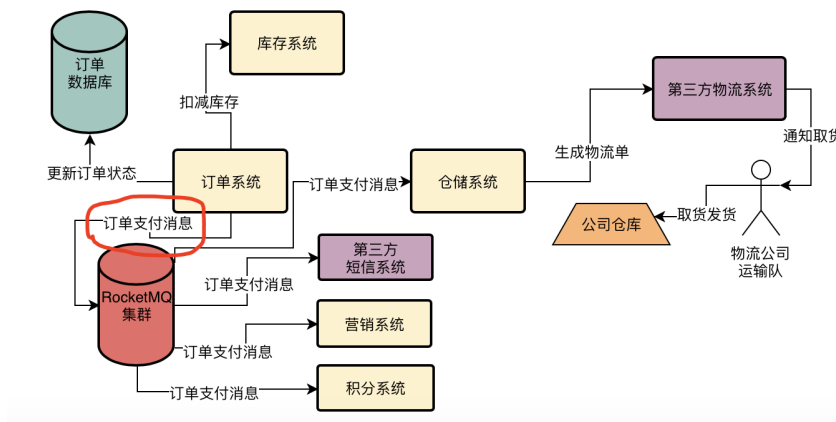
通过上述代码，积分系统、营销系统、推送系统、仓储系统，就可以从RocketMQ里消费“TopicOrderPaySuccess”中的订单消息，然后根据订单消息执行增加积分、发送优惠券、发送短信、通知发货之类的业务逻辑了。

这些系统在RocketMQ部署图中对应的实际上就是下图中画圈的消费者部分。

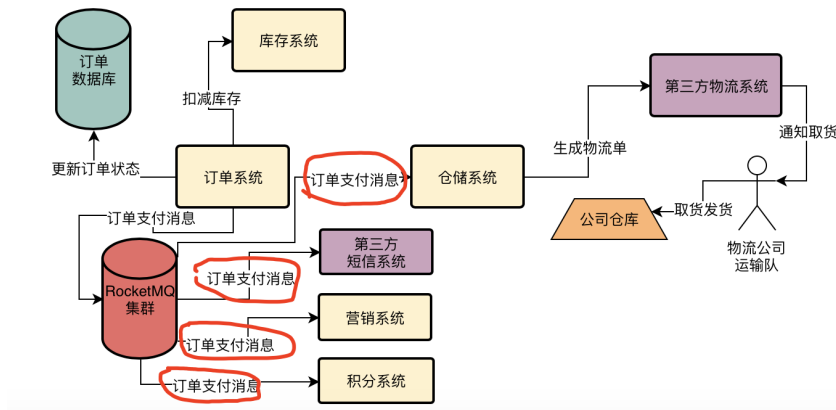


7、订单核心流程改造的流程梳理

接着我们来做一个小小的总结，当各个系统都落地该方案之后，并且部署上线之后，订单系统就会如下图红圈所示，每次支付成功后仅仅更新自己的订单状态，同步扣减库存，接着就会发送消息到RocketMQ里去。



然后推送系统、营销系统、积分系统、仓储系统一旦部署了改造后的代码，就会如下图红圈所示，从RocketMQ里不停的获取订单消息并且执行对应的业务逻辑。



通过上述改造，可以将订单核心流程的性能从1秒~几秒的情况优化到100ms+，可以实现10倍性能提升的效果。

End

专栏版权归公众号**狸猫技术窝**所有

未经许可不得传播，如有侵权将追究法律责任

狸猫技术窝其他**精品专栏**推荐：

[《从零开始带你成为JVM实战高手》](#)

[《21天Java 面试突击训练营》（分布式篇）](#)（现更名为：[互联网Java工程师面试突击第2季](#)）

[互联网Java工程师面试突击（第1季）](#)


重要说明:

如何提问: 每篇文章都有评论区, 大家可以尽情在评论区留言提问, 我会逐一答疑

如何加群: 购买了狸猫技术窝专栏的小伙伴都可以加入**狸猫技术交流群**

具体加群方式, 请参见**目录菜单**下的文档: 《付费用户如何加群? 》**(购买后可见)**

Copyright © 2015-2019 深圳小鹅网络技术有限公司 All Rights Reserved. 粤ICP备15020529号

 小鹅通提供技术支持